Immagine che contiene testo

Descrizione generata automaticamente

**UNIVERSIT****À** DEGLI **STUDI**

DEL **SANNIO**

Benevento - A.A 2020/21

Dynamic Client Framework

>> Corso di Laurea Magistrale in Ingegneria Informatica

>> Architettura e Sistemi Software Distribuiti

>> Progetto 6

>> Gruppo 1: Assunta De Caro, Pietro Vitagliano

Sommario

[ANALISI DEL PROBLEMA 2](#_Toc75770107)

[Obiettivi del Sistema 2](#_Toc75770108)

[Vincoli e soluzioni tecnologiche 2](#_Toc75770109)

[ARCHITETTURA DEL SISTEMA 3](#_Toc75770110)

[DESIGN CLASS DIAGRAM 3](#_Toc75770111)

[Strategie di gestione della congestione 5](#_Toc75770112)

[Gestione della connessione e polling 7](#_Toc75770113)

[Configurazione del sistema 8](#_Toc75770114)

[PRESTAZIONI 8](#_Toc75770115)

[LICENZA E DISTRIBUZIONE 10](#_Toc75770116)

# ANALISI DEL PROBLEMA

## Obiettivi del Sistema

Realizzare un connettore (nodo sensore - piattaforma) che supporti una gestione dinamica della frequenza di raccolta dei dati. L’obiettivo è quello di evitare che si manifestino dei sovraccarichi dal punto di vista delle risorse. Allo scopo è possibile monitorare uno o più parametri prestazionali (es. dimensione delle code) e dimensionare dinamicamente il rate di trasmissione dei nodi sensore, utilizzando diverse politiche (es. aggregazione sul nodo, variazione di frequenza, drop di dati).

## Vincoli e soluzioni tecnologiche

La coda da monitorare è gestita da un broker ActiveMQ Artemis che supporta i seguenti protocolli di comunicazione: JMS, MQTT, OpenWire, HornetQ, AMQP; la soluzione prevista dovrà offrire almeno l’implementazione basata su JMS e MQTT, tuttavia dovrà risultare estendibile nei confronti degli altri protocolli.

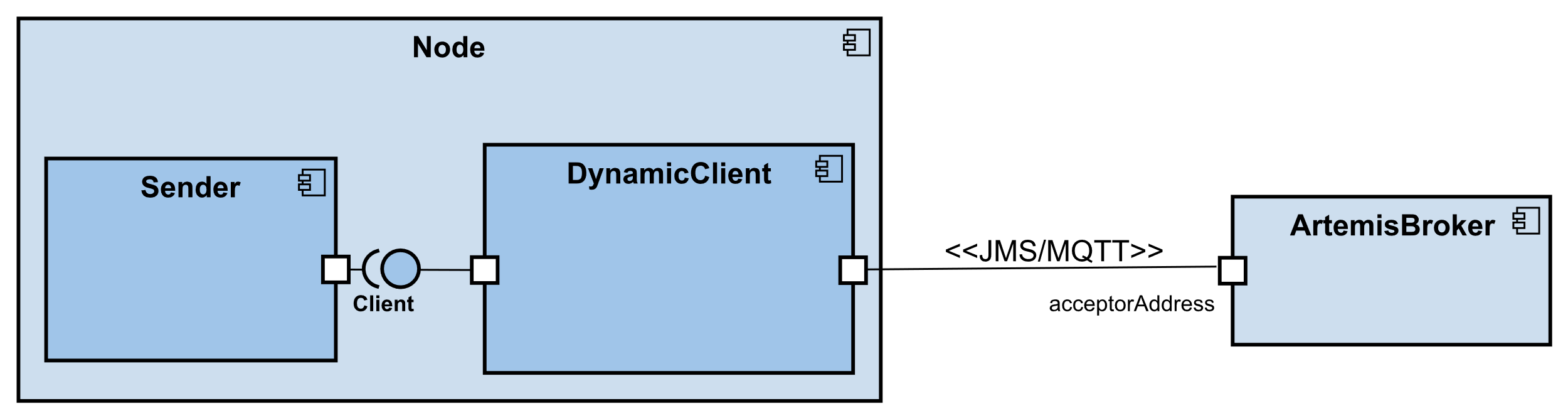
Immagine che contiene testo

Descrizione generata automaticamenteJMS verrà impiegato anche per recuperare le informazioni fondamentali per monitorare il livello di congestione della destinazione e attivare le strategie di gestione. Per recuperare il numero di messaggi non ancora consumati è necessario contattare un indirizzo speciale di sistema, *activemq.management*, specificando il nome della destinazione da monitorare, l’operazione di gestione e i parametri. La documentazione di ArtemisMQ mette a disposizione tutti i dettagli per la connessione.

Ricordiamo, inoltre, che è possibile implementare lo stesso meccanismo anche utilizzando JMX.

# ARCHITETTURA DEL SISTEMA

La soluzione proposta corrisponde alla progettazione di un **framework OO black box** e configurabile dall’utente tramite un file di properties esterno, al fine garantire il tuning dei valori di default ove fosse necessario.

Il Dynamic Client dovrà incapsulare sia i meccanismi di comunicazione, sia quelli de gestione del rate di trasmissione, rendendoli totalmente trasparenti al nodo sensore e offrendogli un’interfaccia minimale per l’invio dei messaggi.

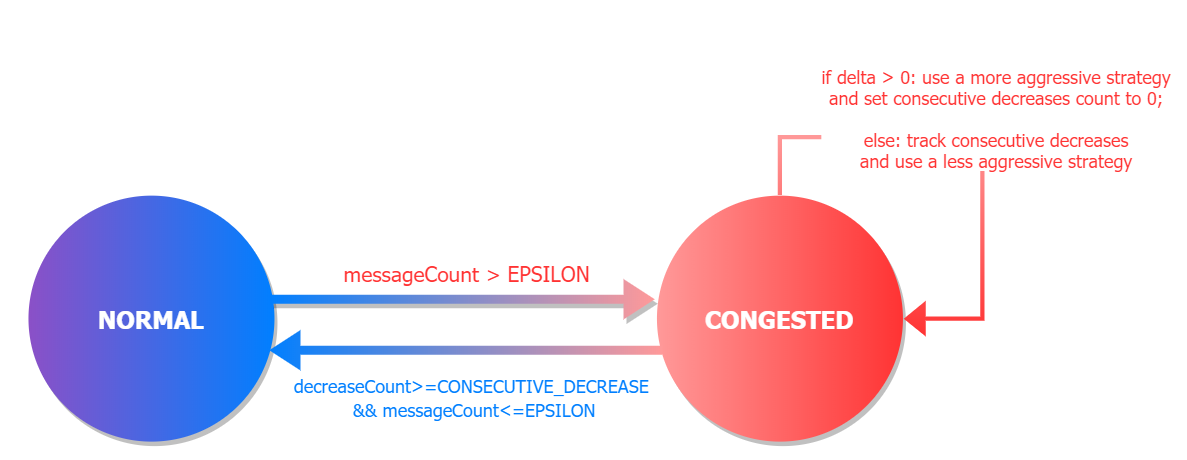
# DESIGN CLASS DIAGRAM

|  |  |
| --- | --- |
| Client | É l’interfaccia esposta dal Dynamic Framework, implementata da Dynamic Client. Prevede i metodi di avvio/terminazione del client e invio dei messaggi. |
| Dynamic Client | Classe astratta che fattorizza la gestione delle strategie e l’aggiornamento dello stato della coda, rendendoli uniformi per tutte le sue implementazioni concrete. Implementa l’interfaccia Client. Delega la scrittura dei metodi *sendMessage*, *startConnection*, *stopConnection* alle sue sottoclassi perché dipendenti dal protocollo di comunicazione utilizzato. |
| Dynamic JMS Client | Classe concreta che estende Dynamic Client, incapsulando gli oggetti necessari a gestire uno scambio dati basato su protocollo JMS. |
| Dynamic MQTT Client | Classe concreta che estende Dynamic Client, incapsulando gli oggetti necessari a gestire uno scambio dati basato su protocollo MQTT. |
| Dynamic Client Creator | Per nascondere la creazione di uno specifico Client e ridurre l’impatto di un cambiamento sul nodo utilizzatore viene impiegato il pattern Creator. Questa è l’interfaccia della factory che delinea il metodo *create*. Le uniche informazioni necessarie alla creazione saranno il nome della coda/topic e l’indirizzo dell’accettore del broker. |
| DJMS Client Creator | Factory concreta di un Dynamic JMS Client. |
| DMQTT Client Creator | Factory concreta di un Dynamic MQTT Client. |
| Polling Service | Recupera le informazioni sullo stato della coda e aggiorna periodicamente il Dynamic Client. L’interazione è basata su JMS. |
| Sample | Un oggetto di tipo Client accetta l’inoltro di un Sample, che incapsula il payload del messaggio e richiede un Time To Live, scaduto il quale verrà considerato non valido per l’invio. Implementa l’interfaccia Serializable. |

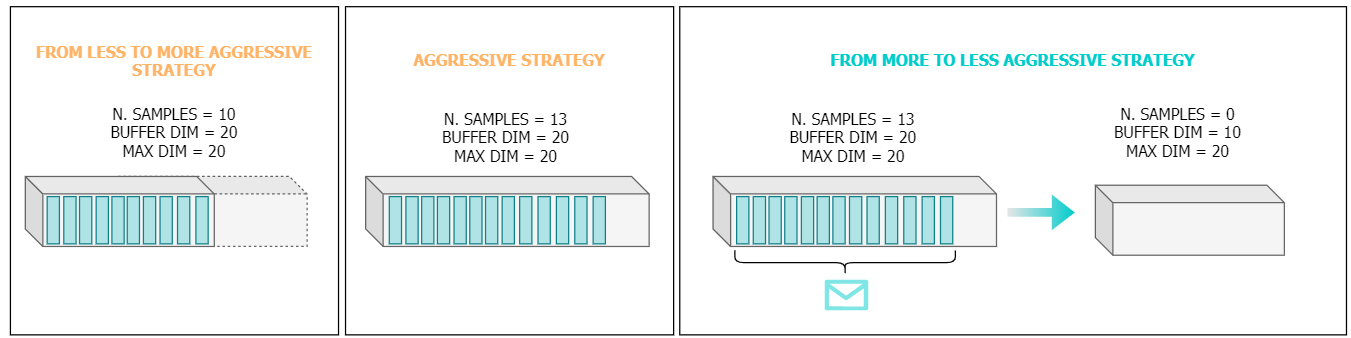
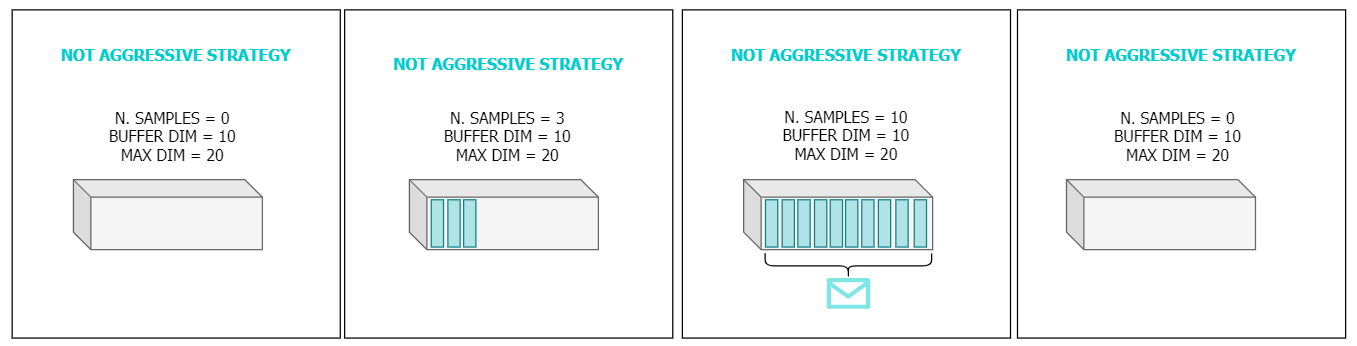
## Strategie di gestione della congestione

Con riferimento alla classe Dynamic Client, andiamo a descrivere le scelte implementative adottate per la sua realizzazione.

Lo stato della coda può essere normale o congestionato. Si permane nel primo fin quando l’incremento dei messaggi nella coda non supera una soglia fissata. Per calcolare l’incremento basta confrontare il numero dei messaggi attuali con quello restituito al check precedente.

Quando ci si trova nello stato di congestione, si applica la strategia selezionata (drop o aggregable) e si monitora la loro variazione. In entrambi i casi, si ricorre a un buffer in cui memorizzare i sample e differire il loro invio, abbassando il rate del sender. Se la variazione è positiva, poiché la coda si sta riempiendo, si aumenta l’aggressività della strategia, ampliando il buffer. Quando si verificano un certo numero di decrementi consecutivi, si riduce l’aggressività della strategia, e, se il numero di messaggi nella coda è minore della soglia indicata, si ritorna nello stato normale.

Il comportamento del buffer merita una spiegazione dettagliata. All’avvio il buffer è vuoto, la sua dimensione minima è fissata dall’utente nel file di configurazione. Con l’aggressività della strategia manipoliamo il numero massimo di sample ammissibili al suo interno. Aumentandola raddoppiamo la grandezza del buffer, al contrario, ridurre l’aggressività significa dimezzarla. Quando il buffer si riempie bisogna svuotarlo: l’aggregazione prevede l’invio di un messaggio che corrisponde al valore calcolato applicando ai sample validi un’operazione tra quelle possibili (di media, massimo, minimo o somma). Con il drop ci limitiamo a inviare i campioni validi, essendo rivolto a dati non aggregabili. Nella figura sottostante, due possibili evoluzioni del buffer: una rappresentativa del ciclo di riempimento/svuotamento in condizioni invariate, l’altra descrivente l’estensione e la contrazione del buffer al cambiamento dell’aggressività.



Nello stato normale si procede direttamente con l’invio del sample corrente e di quelli eventualmente rimasti all’interno del buffer. A prescindere dalla situazione, se il buffer è riempito per più della metà, verrà inviata solo una parte dei sample validi, per evitare una scarica eccessiva che potrebbe generare o peggiorare la congestione. I rimanenti verranno inoltrati all’invio del prossimo sample. Un sample è ritenuto valido finché non è scaduto il suo time to live.

I metodi attivamente coinvolti nella gestione della congestione e la cui logica è definita in Dynmic Client sono i seguenti:

|  |  |
| --- | --- |
| TrySending | Prova ad inviare il sample accettato come parametro controllando lo stato della congestione. Se è NORMAL, procede con l’invio ed eventuale svuotamento del buffer, altrimenti richiama il metodo *handleStrategy* per applicare la strategia richiesta dall’utente. |
| HandleStrategy | A seconda della strategia richiama i metodi *drop* o *aggregable*. |
| Drop | Se il buffer non è pieno, aggiunge il sample per differirne l’invio, altrimenti provvede prima a svuotarlo. |
| Aggregate | Se il buffer non è pieno provvede a memorizzare il sample in attesa che si riempia, si possa effettuare l’aggregazione sui soli campioni validi e, al temine, si elimini il suo contenuto. Richiama *computeAggregation*. |
| Compute  Aggregation | Ritorna il valore ottenuto dall’aggregazione a seconda dell’operazione indicata dall’utente (minimo, massimo, somma, media). Se non ci sono valori validi ritorna un oggetto nullo. |
| Update  QueueStatus | Aggiorna lo stato della coda in base al valore dei messaggi contenuti in essa e alla variazione rispetto al controllo precedente. Infine, aggiorna la capacità del buffer a seconda del livello di aggressività, modellando un diverso comportamento in base alla strategia. |
| EmptyBuffer | Svuota il buffer; se il numero di campioni è maggiore alla metà della capacità, ne invia solo una parte, altrimenti tutti. Ovviamente, i sample inviati devono essere validi. |

## Gestione della connessione e polling

Il Dynamic Client deve concretizzare i metodi richiesti dall’interfaccia Client relativi all’avvio e alla terminazione del servizio. L’operazione di *startClient* si occupa di avviare il servizio di polling, invocando il metodo *startPolling* sull’istanza di PollingService, ed eseguire eventuali operazioni preliminari per la comunicazione, le quali differiscono a seconda del protocollo e sono delegate alle sue sottoclassi. Pertanto, i metodi *startConnection* e *stopConnection* saranno dichiarati astratti e protected. DynamicJMSClient, ad esempio, provvederà a creare ed avviare una connessione, una sessione e un sender. Analogamente, DynamicMQTTClient inizializzerà un publisher e una connessione.

Il polling verrà eseguito da un thread a cadenza regolare, richiamando sull’istanza del Dynamic Client il metodo di aggiornamento dello stato, *updateQueueStatus*, dopo aver recuperato le informazioni sulla destinazione. Ricordiamo che PollingService sfrutta un connettore JMS per contattare il broker.

Dualmente a quanto descritto, il metodo *stopClient* svuota definitivamente il buffer, richiama *stopPolling*, ed esegue le operazioni necessarie a terminare la comunicazione.

## Configurazione del sistema

All’interno del folder Resources è possibile accedere al file “config.properties” che contiene le proprietà modificabili dall’utente per adattarle a contesti specifici.

|  |  |  |
| --- | --- | --- |
| PROPRIETÀ | DESCRIZIONE | VALORE DI DEFAULT |
| epsilon | Soglia di default, superata la quale si segnala una congestione. | 5 |
| consecutive  Decreases | Numero di decrementi consecutivi necessari a ridurre l’aggressività della strategia; abbinato ad un numero di messaggi in coda inferiore ad epsilon, permette di tornare nello stato normale. | 5 |
| bufferDim | Grandezza minima di partenza del buffer. | 10 |
| pollingPeriod | Periodo di polling per aggiornare lo stato della coda. Espresso in ms. | 1000 |
| strategy | Strategia scelta per gestire la congestione: drop o aggregable. | AGGREGABLE |
| aggregationType | Operazione di aggregazione nel caso sia selezionata l’omonima strategia. È possibile optare per: mean, min, max, sum. | MEAN |
| TTL | Time to live da assegnare ai messaggi che contengono il valore computato tramite l’aggregazione. Espresso in ms. | 2000 |

# PRESTAZIONI

Per verificare il funzionamento del framework è stata implementata una versione di test, sia JMS che MQTT, che consente di simulare un’interazione e di monitorarla dal punto di vista di un nodo. È possibile utilizzare il codice come riferimento per l’implementazione. La frequenza di consumo e trasmissione è generata casualmente entro un range da 500 a 3000 ms. È stata introdotta una classe Metric per tenere traccia del numero di congestioni in una simulazione, il periodo di congestione più duraturo, il numero medio di messaggi in coda, il tempo medio di congestione totale.

Per ciascuna strategia sono state effettuate 10 simulazioni da 1 min con client JMS, prima con un rapporto tra sender/consumer di 1:1, poi di 2:1. Il file di configurazione mantiene i valori di default.

TCT = Total Congestion Time

TNT = Total Normal Time

LCP = Longest Congestion Time

MQM = Mean Queued Messages

TCT/ST = Total Congestion Time / Simulation Time

DROP 5 senders / 5 consumer

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | Congestion Times | TCT (s) | TNT(s) | LCP (s) | MQM | TCT/ST (%) |
| 1 | 6 | 27,11 | 32,89 | 8,12 | 3,80 | 45,18 |
| 2 | 0 | 0,00 | 60,00 | 0,00 | 6,93 | 0,00 |
| 3 | 2 | 6,14 | 53,86 | 3,09 | 2,40 | 10,23 |
| 4 | 5 | 22,38 | 37,62 | 10,21 | 3,38 | 37,31 |
| 5 | 4 | 10,65 | 49,35 | 3,05 | 3,40 | 17,76 |
| 6 | 8 | 27,34 | 32,66 | 5,08 | 2,83 | 45,57 |
| 7 | 1 | 3,04 | 56,96 | 3,04 | 0,67 | 5,07 |
| 8 | 4 | 12,17 | 47,83 | 3,06 | 2,22 | 20,29 |
| 9 | 0 | 0,00 | 60,00 | 0,00 | 1,30 | 0,00 |
| 10 | 5 | 15,19 | 44,81 | 3,05 | 2,62 | 25,32 |
| MEAN | 3,50 | 12,40 | 47,60 | 3,87 | 2,95 | 20,67 |

DROP 10 senders / 5 consumer

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | Congestion Times | TCT (s) | TNT(s) | LCP (s) | MQM | TCT/ST (%) |
| 1 | 5 | 30,37 | 29,63 | 16,20 | 4,60 | 50,61 |
| 2 | 6 | 42,53 | 17,47 | 27,35 | 6,55 | 70,89 |
| 3 | 6 | 27,34 | 32,66 | 12,16 | 3,48 | 45,57 |
| 4 | 5 | 18,25 | 41,75 | 6,08 | 3,62 | 30,42 |
| 5 | 6 | 38,43 | 21,57 | 22,28 | 5,35 | 64,05 |
| 6 | 4 | 20,27 | 39,73 | 10,15 | 3,20 | 33,78 |
| 7 | 5 | 27,95 | 32,05 | 16,25 | 4,63 | 46,58 |
| 8 | 4 | 25,29 | 34,71 | 15,18 | 3,47 | 42,15 |
| 9 | 5 | 21,26 | 38,74 | 9,12 | 3,07 | 35,43 |
| 10 | 4 | 33,46 | 26,55 | 21,30 | 4,82 | 55,76 |
| MEAN | 5,00 | 28,51 | 31,49 | 15,61 | 4,28 | 47,52 |

AGGREGABLE 5 senders / 5 consumer

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | Congestion Times | TCT (s) | TNT(s) | LCP (s) | MQM | TCT/ST (%) |
| 1 | 1 | 3,04 | 56,96 | 3,04 | 2,87 | 5,07 |
| 2 | 4 | 12,15 | 47,85 | 3,05 | 3,07 | 20,25 |
| 3 | 6 | 18,20 | 41,80 | 3,05 | 3,55 | 30,33 |
| 4 | 6 | 15,76 | 44,24 | 3,05 | 3,00 | 26,26 |
| 5 | 4 | 14,21 | 45,79 | 5,08 | 2,33 | 23,68 |
| 6 | 6 | 22,40 | 37,60 | 5,07 | 3,80 | 37,33 |
| 7 | 4 | 14,19 | 45,82 | 5,09 | 3,13 | 23,64 |
| 8 | 7 | 19,56 | 40,444 | 3,05 | 2,92 | 32,59 |
| 9 | 6 | 23,31 | 36,69 | 8,15 | 3,1 | 38,84 |
| 10 | 7 | 19,88 | 40,12 | 3,05 | 2,68 | 33,13 |
| MEAN | 5,1 | 16,27 | 43,73 | 4,17 | 3,04 | 27,11 |

AGGREGABLE 10 senders / 5 consumer

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | Congestion Times | TCT (s) | TNT(s) | LCP (s) | MQM | TCT/ST (%) |
| 1 | 1 | 32,45 | 27,55 | 32,45 | 5,67 | 54,09 |
| 2 | 3 | 16,18 | 43,82 | 10,11 | 3,90 | 26,97 |
| 3 | 4 | 23,29 | 36,71 | 14,15 | 4,00 | 38,81 |
| 4 | 1 | 16,20 | 43,80 | 16,20 | 3,73 | 27,00 |
| 5 | 4 | 29,37 | 30,63 | 19,23 | 2,95 | 48,95 |
| 6 | 3 | 18,22 | 41,78 | 12,15 | 3,77 | 30,37 |
| 7 | 6 | 22,25 | 37,75 | 5,07 | 2,87 | 37,09 |
| 8 | 3 | 19,24 | 40,76 | 11,17 | 4,10 | 32,06 |
| 9 | 5 | 28,33 | 31,67 | 16,17 | 4,20 | 47,22 |
| 10 | 1 | 11,14 | 48,86 | 11,14 | 3,60 | 18,57 |
| MEAN | 3,10 | 21,67 | 38,33 | 14,78 | 3,88 | 36,11 |

Per ciascuna strategia, l’aumento del rapporto sender/consumer si riflette anche su un peggioramento delle prestazioni per via dell’aumento della pressione; tuttavia, non si trascorre mai più della metà del tempo in situazione di congestione, dimostrando l’efficacia del client. Con un numero superiore di sender, crescerà anche il numero medio di messaggi in coda, mantenendosi sempre sotto la soglia di congestione. Si evince anche un maggior numero di volte in cui si verifica congestione non si traduce necessariamente in periodi molto lunghi di recovery: se osserviamo il report di Aggregable possiamo notare una diminuzione del CongestionTimes ma aumento del tempo totale passato in quello stato, nonostante la crescita del rapporto sender/consumer. In entrambi i casi è evidente che la strategia di aggregazione restituisce una performance nel complesso migliore, con il 36% del TCT contro il 27% del drop, un numero medio di messaggi e congestioni lievemente inferiore. Ciò è dovuto alla possibilità di svuotare il buffer più efficientemente in situazioni di stress, ricorrendo alla sintesi, e non all’invio di più messaggi.

# LICENZA E DISTRIBUZIONE

La licenza scelta per il progetto è la GPL v.3, riportata nel root. È possibile trovarlo su GitHub con la sua release al seguente indirizzo: <https://github.com/assuntaDC/DynamicClientFramework.git>.